Making Fortran programs for solving an advection equation using Upwind and Lax-Wendroff methods

Written by Evgenii Neumerzhitckii

Sep 18, 2019

Contents

1 Reorganizing Fortran code

1.1 Creating a Makefile

We create a Makefile based on the post by a Stackoverflow user Sophie:

`https://stackoverflow.com/a/30142139/297131`.

A useful feature of this Makefile is that it places all the build files (*.o, *.mod and executables) into a separate directory called `build`. This is, arguably, a better file organization than mixing both the source code and binary files in the same directory.

Furthermore, we have extended the Makefile by adding ability to create an executable for unit tests by running

```
$ make test
```

command. This is accomplished by selecting files that have names ending with "`_test.f90`" and using them to build the test executable. When the main executable is made, those test source files are excluded.

1.2 Creating `Output` module

We create an `Output` module. It contains `write_output` function that writes x, t values, as well as the solution 2D array, to a binary data file. The format of the data file is documented in the README.md. We have chosen a binary file instead of a text file because a binary file needs less storage space, and it is faster to read and write.

1.3 Creating `Grid` module

Next, we write a `Grid` module that initializes the arrays for storing x, t and solution values.

1. Array of x values

First, we create a 1D array that stores the x values. The range for the x variable is divided into cells of equal size located between x_start and x_end numbers. The total number of cells is set by the nx parameter. Parameters nx, x_start and x_end are supplied to the program by the user. The x values are set to be in the middle of their cells.

For example, suppose we have four x cells, starting from 0 and ending with 10, as shown on Figure 1. In this case the x array will contains four values: 1.25, 3.75, 6.25 and 8.75.

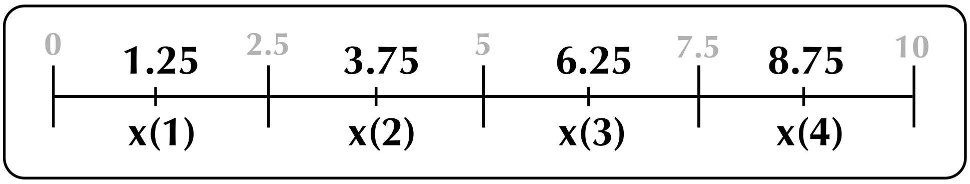

Figure 1: Array for storing four x values that are evenly spread in the middle of their cells between zero and ten.

2. Array of *t* values

Next, we allocate a 1D array to store the time values. The time values start with *t_start* and end with *t_end* that are supplied by the user. However, unlike the x values, the number of t values is not supplied by the user, since, in general, the size of the time step can change during calculation. Therefore, we do not yet know the final size of the t array.

This creates a "chicken and egg" problem: we need to initialize a t array in order to start calculations, but we do not yet know the size of this array, since it will be determined when calculations are completed. This issue is solved by using an "allocatable" keyword that creates a dynamical array, as shown on `Line 46` in Listing 1. On `Line 54` we allocate an arbitrary number of t values. Then, later in our program, when we calculate the solution and need to store more t values, we will make the t array larger.

Listing 1: Initializing a dynamical array for storing time values (`grid.f90`).

```
46 real(dp), allocatable :: t_points(:)
47
48 ! Initial size of the t array, an arbitrary number.
49 ! The array will be enlarged when new solutions are calculated.
50 nt_allocated = 13
51
52 ! Allocate memory for t values.
53 allocate(t_points(nt_allocated))
```

3. Solution array

Finally, we initialize the most important array that will store the values of the solution. This is a 2D array, its first index is the x dimension, and the second is the time dimension. The array will store the solution for every time iteration. This is not be a good idea if the number of time iterations is very large, since the array may become larger than available memory. However, in our case we use fewer than a thousand time iterations, and consequently the solution array will be small relative to the total memory available.

An example of a solution array is shown on Figure 2. Here we create an array for four x values and three t values. In addition, the array contains two extra columns (one on the left and one the right) for storing values of so-called "ghost" cells. These are temporary cells that help to make calculation of the solution simpler. These ghost cells will be removed from the solution array when calculations are finished.

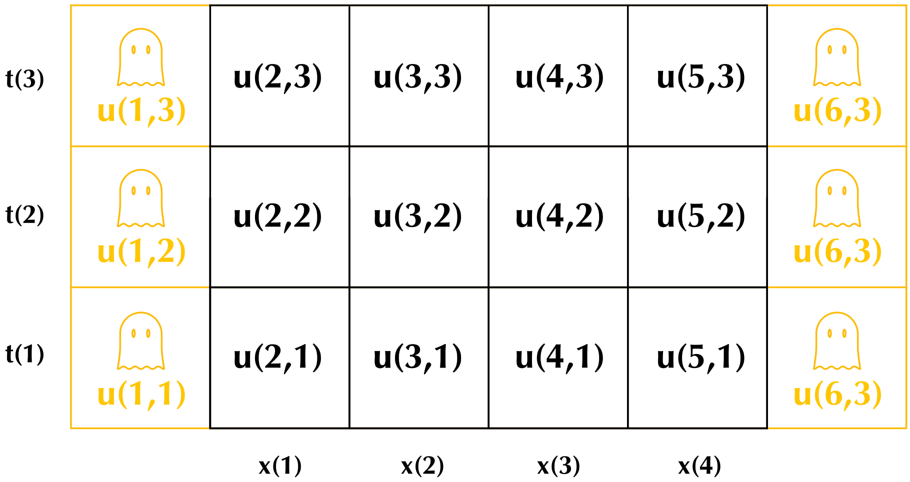

Figure 2: An array for storing solution values for four x values and three t values. The array contains two additional columns at the start and the end for storing values for the "ghost" cells, which are used for making calculations simpler.

The size of the time dimension of the solution array is determined by the number of the t values. Since this number is not known in advance, we use the trick from the time array initialization. Specifically, we create a dynamical array and allocate its memory using an arbitrary number of time values, as shown in Listing 2. Later, as we calculate the solution for growing number of time values, we will resize the solution array and make its time dimension larger.

Listing 2: Initializing a dynamical array for storing values of the solution. The x dimension contains two additional columns for storing the "ghost" cells. The y dimension has an arbitrary size that will be increased later during calculation to store data for new time values. (`grid.f90`).

```
real(dp), allocatable :: solution(:, :)

allocate(solution(nx + 2, nt_allocated))
```

1.4 Creating `InitialConditions` and `Step` module

Next, we create `InitialConditions` and `Step` modules for calculating initial conditions and performing one step of integration.

2 Implementing Upwind scheme

We implement Upwind method and solve advection equation

$$u_t + v\, u_x = 0,$$

with initial condition

$$u(x, 0) = \sin(2\pi x),$$

for x from 0 to 1. A solution for Upwind scheme are shown on Figure 3.

3 Implementing Lax-Wendroff scheme

Next, we implement Lax-Wendroff method. Exact and approximate solutions for Lax, Upwind and Lax-Wendroff methods at $t = 1\ s$ are shown on Figure 3 and a video is available here:

`https://youtu.be/zWhv5VOyjhE`

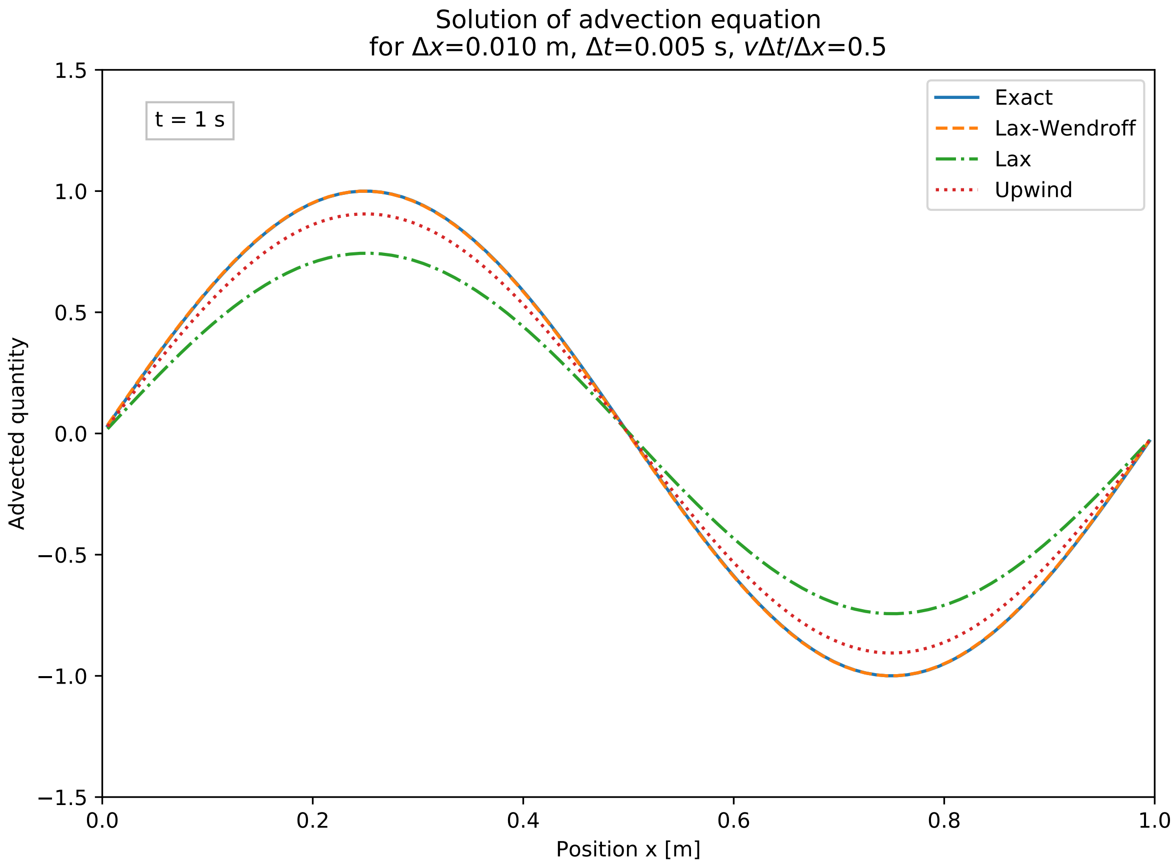

Figure 3: Exact and approximate solutions to advection equation at $t = 1\ s$ with sine initial conditions and courant factor $C = 0.5$.

4 Solutions for square initial conditions

Next, we solve the equation with square initial conditions. Solutions are shown on Figure 4 and a video is available here:

https://youtu.be/7sHFfG8Rf-A

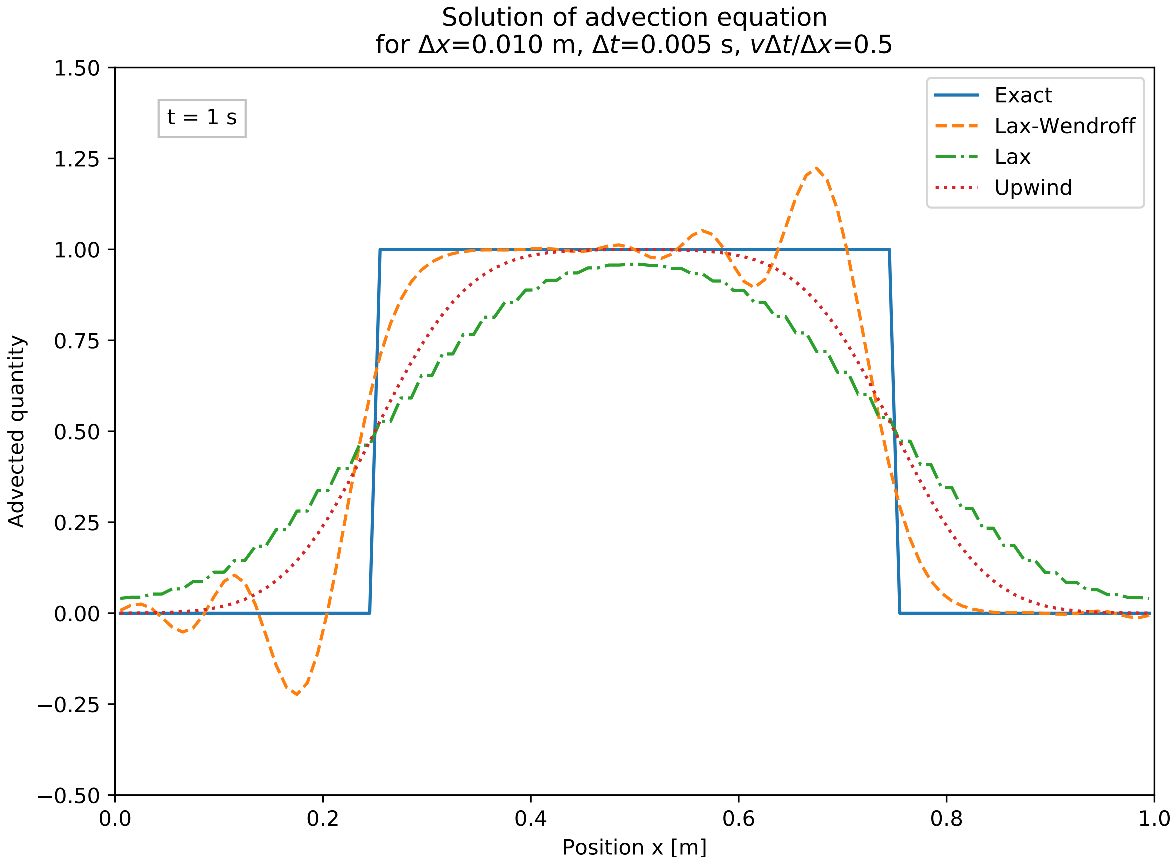

Figure 4: Exact and approximate solutions to advection equation at $t = 1\ s$ with square initial conditions and courant factor $C = 0.5$.

5 Solutions for Courant factor $C = 1$

Next, we use Courant factor $C = 1$ instead of $C = 0.5$ and solve the equation both with sine and square initial conditions.

5.1 Sine initial conditions

Solutions for sine initial conditions and $C = 1$ are shown on Figure 5 and a video is available here:

https://youtu.be/Y47G1ylsle8

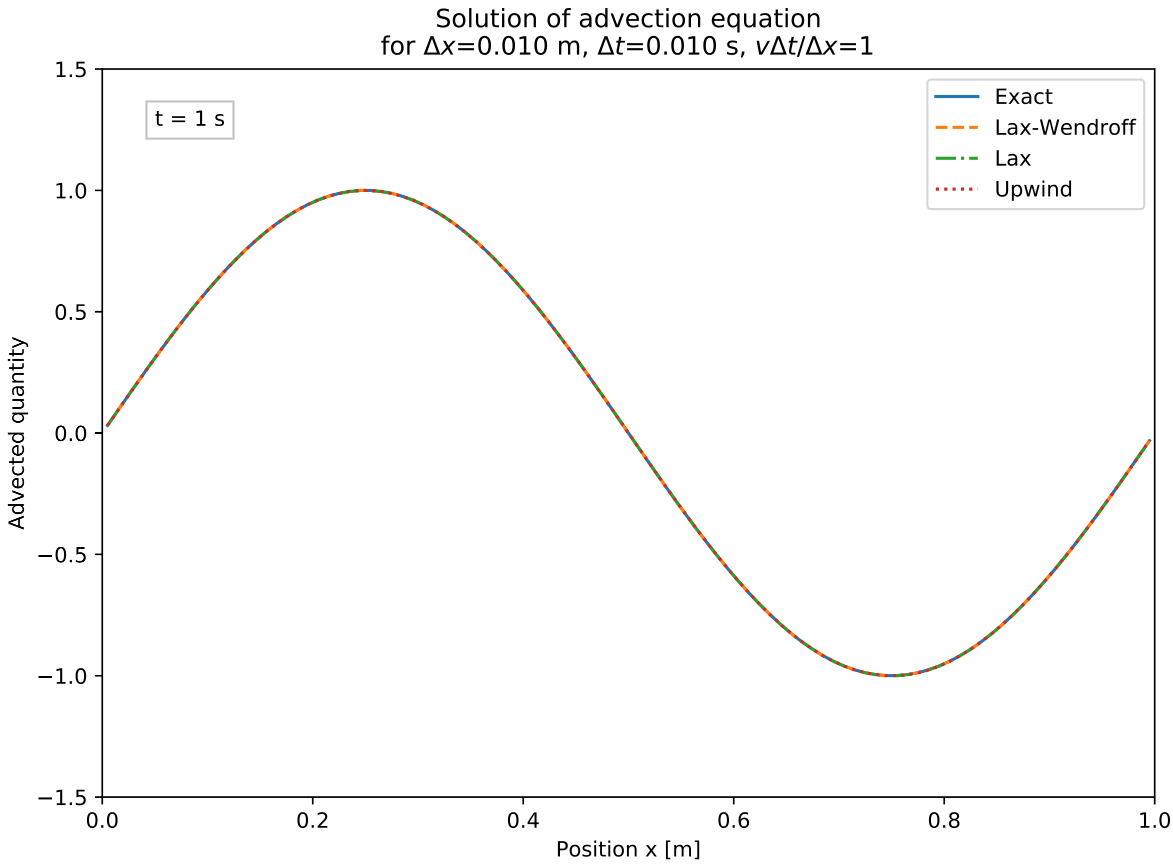

Figure 5: Exact and approximate solutions to advection equation at $t = 1$ s with sine initial conditions and courant factor $C = 1$.

5.2 Square initial conditions

Solutions for square initial conditions and $C = 1$ are shown on Figure 6 and a video is available here:

`https://youtu.be/K3C1n1MkSzQ`

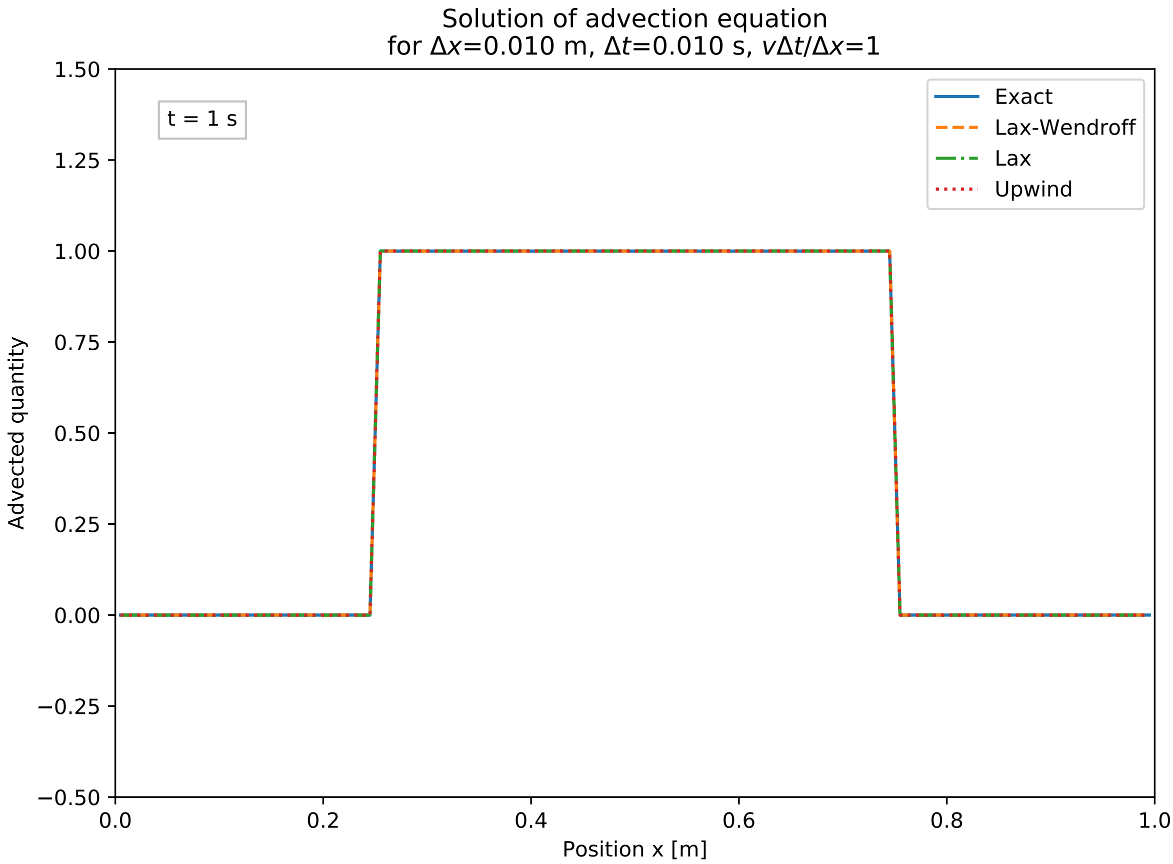

Figure 6: Exact and approximate solutions to advection equation at $t = 1\ s$ with square initial conditions and courant factor $C = 1$.

6 Analysis of results

6.1 Numerical diffusion and dispersion

We can see from videos

`https://youtu.be/zWhv5VOyjhE`

and

`https://youtu.be/7sHFfG8Rf-A`

that solutions from Lax and Upwind methods spread out and deviate from exact solution as time increases. This can be explained by the $\frac{\partial^2 u}{\partial x^2}$ remainder terms in the modified PDEs of the two methods, which produce *numerical diffusion*.

In contrast, Lax-Wendroff method does not have the diffusion term, and we can see that Lax-Wendroff's solution is indistinguishable from exact one for the sine initial conditions (Figure 3).

However, the modified PDE of Lax-Wendroff method includes $\frac{\partial^3 u}{\partial x^3}$ term that generates oscillations near discontinuous regions (also known as *numerical dispersion*), which can be seen on video

`https://youtu.be/7sHFfG8Rf-A`

6.2 Exact solutions for Courant factor $C = 1$

We can see from Figures 5 and 6 that solutions from all three methods are indistinguishable from exact ones for both sine and square initial conditions. This happens probably because the error terms in the modified PDEs of the methods include factors like $(1 - C)$ and $(1 - C^2)$. Consequently, the error terms vanish for $C = 1$ and approximate solutions become exact.